International Journal of Advances in Engineering and Management (IJAEM)
Volume 2, Issue 12, pp: 470-472 www.ijaem.net ISSN: 2395-5252

Database Forensics in Open Source Database

Assistant Professor, Ms. Nishi, Student, RaghavAbrol, Student,VipulKukreja
[Dr. Akhilesh Das Gupta Institute of Technology and Management, Shastri Park, Delhi]

Date of Submission: 15-12-2020 Date of Acceptance: 30-12-2020

ABSTRACT: Despite the importance of databases in virtually all data driven applications, database forensics is still not the thriving topic it ought to be. Many database management systems (DBMSs) structure the data in the form of trees, most notably B +-Trees. Since the tree structure is depending on the characteristics of the INSERT-order, it can be used in order to generate information on later manipulations, as was shown in a previously published approach. In this work we analyse this approach and investigate, whether it is possible to generalize it to detect DELETE-operations within general INSERT-only trees. We subsequently prove that almost all forms of B +-Trees can be constructed solely by using INSERToperations, i.e. that this approach cannot be used to prove the existence of DELETE-operations in the past.

I. INTRODUCTION

Digital forensics has become an important factor in the analysis of incidents in the IT world, be it during an official legal investigation, or solely within an internal analysis. While there are many novel approaches in well-researched areas like network or file forensics, the topic of database forensics, i.e. the analysis of databases in order to detect manipulations and hidden information, has never been in the center of attention, even though it gradually gains importance in the scientific community . Especially considering the proclaimed ”age of data science” this seems like a huge blind spot, as most structured data is stored, and often even processed, in some kind of database. Nevertheless, several approaches for database forensics have been devised in the past. Many of these focus on the extraction of information gathered in log-files or related mechanisms, also including NO-SQL databases .

Other approaches rely on the analysis of internal mechanisms used for guaranteeing ACID-compliance like the transaction mechanism . Especially for the first strategy, there exists a large body of knowledge targeting many different database management systems like Oracle or MS SQL . One major drawback of most of theses approaches, when compared to those targeting internal mechanisms, is that logs are written for the purpose of detecting malicious behavior , thus being a primary target for manipulations themselves. Furthermore, users with administrator privileges do have a lot of possibilities regarding log files. This is not so easy with the utilization of internal mechanisms, as manipulations there can possibly destroy the integrity of the database. Another approach that even works a level of abstraction deeper than the utilization of DBMS-specific internal mechanisms was provided . In this approach, the authors use the structure of the resulting B +-Tree that is used to structure the data inside the DBMS in order to detect certain kinds of information. Still, besides the issue of practicability (see for a practical adoption in logging), the approach also requires a certain insertion order of the elements, they have to be entered in a strictly monotonous order with respect to the primary key. In this work we will thus discuss why relaxing this requirement is not easily done and we prove that almost all tree structures as utilized by the original mechanisms can be constructed by solely using INSERT-statements, thus making the detection of DELETEs impossible using this approach

II. BACKGROUND & RELATED WORK

B-Trees and B + -Trees The B-Tree was originally defined by Bayer [4] as a tree structure where all leaf nodes lie on the same level (balanced tree) and the following properties are obeyed:

Every node except the root has between b 2 and b, the root node between 1 and b elements. The value b is constant and predefined for a given tree (”order” of the tree).

An inner node with d elements possesses d + 1 child nodes

The elements inside nodes are sorted. The difference between the classical B-Tree and the B +-Tree is that all the payload in a B +-Tree resides in the leaf nodes, the inner nodes solely hold pointers in order to allow for searching the tree [5]. Insertion in a B +-Tree works as follows:

- The leaf node where the elements should be placed is identified. If this node contains less thanb elements than the new element is simply added to the internal sorted list of the node. If the leaf already contains b elements,its parent node, for which the lowest element in the second leaf is selected. This is done iteratively, i.e. in case the parent node now contains more than b elements, it needs to be split too. This can be required to be done until the root node is reached, In case this contains b + 1 elements after the insertion, it is split too and a new root node is generated, solely holding the lowest element of the second child node as element. B +-Trees are typically used, with slight adaptions, in database storage engines like InnoDB. For example, since full SELECTs are quite common, the leaf nodes are linked with each other in the form of a list, still, the principles of the tree operations stay the same.

Tree data structure works great in representing computations unambiguously. In this study, the researchers enhanced the existing insert algorithm in B*Tree by introducing an expanded algorithm for inserting key values in B*Tree. This would further delay, if not reduce redistribution and frequency of splitting nodes in B*Tree. The idea of the expanded algorithm is to check for a cousin node within the same level that can accommodate a key value from its nearest sibling. A cascading effect is expected until the nearest sibling of the overflowing node can accommodate a key value with the premise that the nearest siblings of the overflowing node are full. Exploring the potential of this algorithm can really simplify a complex problem. The expanded insert algorithm would ensure that all leaf nodes in the B*Tree would be full and not just 2/3 full before a redistribution process is to be performed. Thus, reducing the nodes used in a B*Tree that would result to a far more favorable priori and posteriori estimates. Many scenarios impose a heavy update load on B-tree indexes in modern databases. A typical case is when B-trees are used for indexing all the keywords of a text field. For example upon the insertion of a new text record (e.g. a new document arrives), a barrage of new keywords has to be inserted into the index causing many random disk I/Os and interrupting the normal operation of the database. The common approach has been to collect the updates in a separate structure and then perform a batch update of the index. This update "freezes" the database. Many applications, however, require the immediate availability of the new updates without any interruption of the normal database operation. In this paper we present a novel online B-tree update method based on a new buffering data structure we introduce - Dynamic Bucket Tree (DBT). The DBT-buffer serves as a differential index for new updates. The grouping of keys in DBT-buffer is based on the longest common prefixes (LCP) of their binary representations. The LCP is used as a measure of the locality of keys to be transferred to the main B-tree. Our online update system does not slow down concurrent user transactions or lead to degradation of search performance. Experiments confirm that our DBT buffer can be efficiently used for online updates of text fields. As such it represents an effective solution to the notorious problem of handling updates to an Inverted Index.

B-tree indexes have been used in a wide variety of computing systems from handheld devices to mainframes and server farms. Over the years, many techniques have been added to the basic design in order to improve efficiency or to add functionality. Examples include separation of updates to structure or contents, utility operations such as non-logged yet transactional index creation, and robust query processing such as graceful degradation during index-to-index navigation. Modern B-Tree Techniques reviews the basics of B-trees and of B-tree indexes in databases, transactional techniques and query processing techniques related to B-trees, B-tree utilities essential for database operations, and many optimizations and improvements. It is intended both as a tutorial and as a reference, enabling researchers to compare index innovations with advanced B-tree techniques and enabling professionals to select features, functions, and tradeoffs most appropriate for their data management challenges.

The simplest solution is to store data in an array and append values when new values come. But if you need to check if a given value exists in the array, then you need to search through all of the array elements one by one and check whether the given value exists. If you are lucky enough, you can find the given value in the first element. In the worst case, the value can be the last element in the array. We can denote this worst case as O(n) in asymptotic notation. This means if your array size is “n,” at most, you need to do “n” number of searches to find a given value in an array.

The easiest solution is to sort the array and use binary search to find the value. Whenever you insert a value into the array, it should maintain order. Searching start by selecting a value from the middle of the array. Then compare the selected value with the search value. If the selected value is greater than search value, ignore the left side of the array and search the value on the right side and vice versa.

The database creates a unique random index (or primary key) for each of the given records and converts the relevant rows into a byte stream. Then, it

stores each of the keys and record byte streams on a B+tree. Here, the random index used as the key for indexing. The key and record byte stream is altogether known as Payload. all records are stored in the leaf nodes of the B+tree and index used as the key to creating a B+tree. No records are stored on non-leaf nodes. Each of the leaf nodes has reference to the next record in the tree. A database can perform a binary search by using the index or sequential search by searching through every element by only traveling through the leaf nodes.If no indexing is used, then the database reads each of these records to find the given record. When indexing is enabled, the database creates three B-trees for each of the columns in the table

III. CONCLUSION

Databases should have an efficient way to store, read, and modify data. B-tree provides an efficient way to insert and read data. In actual Database implementation, the database uses both B-tree and B+tree together to store data. B-tree used for indexing and B+tree used to store the actual records. B+tree provides sequential search capabilities in addition to the binary search, which gives the database more control to search non-index values in a database. In this work we showed that approach for B +-Tree-forensics for databases as defined in [12] cannot be generalized for the detection of data deletion in case of a table that allows general, non-monotonous, data insertion. Of course, the approach can still be used for the task it was originally intended, indicating manipulations in Audit & Control tables. It must be noted though that, similar to the original approach, we solely concentrated on the structure of the leaf nodes and did not consider the whole structure of the tree including internal nodes. Still, in essence the techniques work similar for changing the inner node structure, just the amount of elements required to be inserted later gets rather large. Furthermore, these proofs can be trivially expanded from B +-Trees to B $*$ -Trees, as the only difference between the two is the minimal number of elements inside the leaf nodes, the other requirements remain the same, which also means that the proofs given in this paper work similar. Still, since B $*$ -Trees are, to the best of our knowledge, not that relevant in database forensics, and were also not part of the original approach, we skipped the details on them.

ACKNOWLEDGMENT

The writers are very grateful to Dr. Akhilesh Das Gupta Institute of Technology and Management, Delhi, India, for providing eminent computation amenities in the College campus. Authors would also like to pay regards to the Director of College, Department HOD and colleagues for giving their ethical guide and assist in this research work..

REFERENCES

[1]. WK Hauger and MS Olivier. 2018. NoSQL databases: forensic attribution implications. SAIEE Africa Research Journal 109, 2 (2018), 119–132

[2]. Werner K Hauger and Martin S Olivier. 2015. The state of database forensic research. In Information Security for South Africa (ISSA), 2015. IEEE, 1–8

[3]. Peter Kieseberg, Sebastian Schrittwieser, Lorcan Morgan, Martin Mulazzani, Markus Huber, and Edgar Weippl. 2013. Using the structure of b+-trees for enhancing logging mechanisms of databases. International Journal of Web Information Systems 9, 1 (2013), 53–68

[4]. .Peter Kieseberg, Sebastian Schrittwieser, Martin Mulazzani, Markus Huber, and Edgar Weippl. 2011. Trees cannot lie: Using data structures for forensics purposes. In Intelligence and Security Informatics Conference (EISIC), 2011 European. IEEE, 282–285

[5]. . David Litchfield. 2007. Oracle forensics part 1: Dissecting the redo logs. NGSSoftware Insight Security Research (NISR), Next Generation Security Software Ltd, Sutton (2007).

[6]. Gerome Miklau, Brian Neil Levine, and Patrick Stahlberg. 2007. Securing history: Privacy and accountability in database systems.. In CIDR. Citeseer, 387–396.

[7]. TanushreeShelare and VarshaPowar. [n. d.]. A Database Forensic Approach to Detect Tamper Using B+-Trees. ([n. d.]).

[8]. Patrick Stahlberg, Gerome Miklau, and Brian Neil Levine. 2007. Threats to privacy in the forensic analysis of database systems. In Proceedings of the 2007 ACM SIGMOD international conference on Management of data. ACM, 91–102.

[9]. Erin Toombs. 2015. Microsoft SQL server forensic analysis. Ph.D. Dissertation. Utica College.